Separation Predicates: a Taste of Separation Logic in First-Order Logic

François Bobot, Jean-Christophe Filliâtre

Separation Predicates: a Taste of Separation Logic in First-Order Logic*

François Bobot and Jean-Christophe Filliâtre

LRI, Univ Paris-Sud, CNRS, Orsay F-91405
INRIA Saclay-Île-de-France, ProVal, Orsay F-91893

Abstract. This paper introduces *separation predicates*, a technique to reuse some ideas from separation logic in the framework of program verification using a traditional first-order logic. The purpose is to benefit from existing specification languages, verification condition generators, and automated theorem provers. Separation predicates are automatically derived from user-defined inductive predicates. We illustrate this idea on a non-trivial case study, namely the composite pattern, which is specified in C/ACSL and verified in a fully automatic way using SMT solvers Alt-Ergo, CVC3, and Z3.

1 Introduction

Program verification has recently entered a new era. It is now possible to prove rather complex programs in a reasonable amount of time, as demonstrated in recent program verification competitions [17, 12, 10]. One of the reasons for this is tremendous progress in automated theorem provers. SMT solvers, in particular, are tools of choice to discharge verification conditions, for they combine full first-order logic with equality, arithmetic, and a handful of other theories relevant to program verification, such as arrays, bit vectors, or tuples. Notable examples of SMT solvers include Alt-Ergo [4], CVC3 [1], Yices [9], and Z3 [8].

Yet, when it comes to verifying programs involving pointer-based data structures, such as linked lists, trees, or graphs, the use of traditional first-order logic to specify, and of SMT solvers to verify, shows some limitations. Separation logic [22] is then an elegant alternative. Designed at the turn of the century, it is a program logic with a new notion of conjunction to express spatial separation. Separation logic requires dedicated theorem provers, implemented in tools such as Smallfoot [2] or VeriFast [13, 15]. One drawback of such provers, however, is to either limit the expressiveness of formulas (*e.g.* to the so-called symbolic heaps), or to require some user-guidance (*e.g.* open/close commands in Verifast).

In an attempt to conciliate both approaches, we introduce the notion of *separation predicates*. The idea is to introduce some ideas from separation logic into a traditional verification framework where the specification language, the

* This work was (partially) supported by the Information and Communication Technologies (ICT) Programme as Project FP7-ICT-2009-C-243881 CerCo and by the U3CAT project (ANR-08-SEGI-021) of the French national research organization.

verification condition generator, and the theorem provers were not designed with separation logic in mind. Separation predicates are automatically derived from user-defined inductive predicates, on demand. Then they can be used in program annotations, exactly as other predicates, *i.e.*, without any constraint. Simply speaking, where one would write $P \star Q$ in separation logic, one will here ask for the generation of a separation predicate *sep* and then use it as $P \wedge Q \wedge sep(P, Q)$.

We have implemented separation predicates within Frama-C's plug-in Jessie for deductive verification [21]. This paper demonstrates the usefulness of separation predicates on a realistic, non-trivial case study, namely the composite pattern from the VACID-0 benchmark [20]. We achieve a fully automatic proof using three existing SMT solvers.

This paper is organized as follows. Section 2 gives a quick overview of what separation predicates are, using the classic example of list reversal. Section 3 formalizes the notion of separation predicates and briefly describes our implementation. Then, Section 4 goes through the composite pattern case study. Section 5 presents how this framework can be extended to express the set of pointers modified by a function. We conclude with related work in Section 6.

2 Motivating Example

As an example, let us consider the classic in-place list reversal algorithm:

$rev(p) \equiv$
  $q := \texttt{NULL}$
  while $p \neq \texttt{NULL}$ do $t := p{\rightarrow}\texttt{next}$; $p{\rightarrow}\texttt{next} := q$; $q := p$; $p := t$ done
  return q

We may want to verify that, whenever p points to a finite singly-linked list, then $rev(p)$ returns a finite list. (Proving that lists are indeed reversed requires more space than available here.) To do so, we first define the notion of finite singly-linked lists, for instance using the following inductive predicate *islist*:

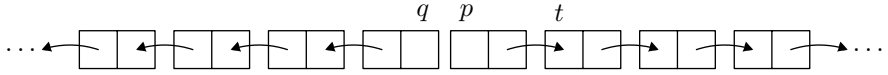
$$\begin{array}{l}\textsf{inductive } islist(p) \equiv \\ \quad \mid C_0 : islist(\texttt{NULL}) \\ \quad \mid C_1 : \forall p.\, p \neq \texttt{NULL} \Rightarrow islist(p{\rightarrow}\texttt{next}) \Rightarrow islist(p)\end{array}$$

Then we specify function *rev* using the following Hoare triple:

$$\{islist(p)\}\ q := rev(p)\ \{islist(q)\}$$

To perform the proof, we need a loop invariant. A natural invariant expresses that both p and q are finite lists, that is $islist(p) \wedge islist(q)$.

Unfortunately, this is not enough for the proof to be carried out. Indeed, we lack the crucial information that assigning $p{\rightarrow}\texttt{next}$ will not modify lists q and t. Therefore, we cannot prove that the invariant above is preserved.



Separation logic proposes an elegant solution to this problem. It introduces a new logical connective $P \star Q$ that acts as the conjunction $P \wedge Q$ and expresses spatial separation of P and Q at the same time. In the list reversal example, it is used at two places. First, it is used in the definition of *islist* to express that the first node of a list is disjoint from the remaining nodes:

$$islist(p) \;\equiv\; \mathsf{if}\ p = \texttt{NULL}\ \mathsf{then}\ \mathsf{emp}\ \mathsf{else}\ \exists q.\, p{\rightarrow}\texttt{next} \mapsto q \star islist(q)$$

This way, we can now prove that list t is preserved when $p{\rightarrow}\texttt{next}$ is assigned. Second, the connective $\star$ is also used in the loop invariant to express that lists p and q do not share any pointer:

$$islist(p) \star islist(q).$$

This way, we can now prove that list q is preserved when $p{\rightarrow}\texttt{next}$ is assigned. Using a dedicated prover for separation logic, list reversal can be proved correct using this loop invariant.

In our attempt to use traditional SMT solvers instead, we introduce the notion of *separation predicates*: the $\star$ connective of separation logic is replaced by new predicate symbols, which are generated on a user-demand basis. Our annotated C code for list reversal using separation predicates is given in Fig. 1.

We define predicate `islist` inductively (lines 4–8), as we did earlier in this section. In this definition `\valid(p)` express that `p` is a pointer that can be safely dereferenced (allocated and not freed). It captures finite lists only and, consequently, the first node of a list is disjoint from the remaining nodes. However, such a proof requires induction and thus is out of reach of SMT solvers. We add this property as a lemma (lines 11–12), using a separation predicate `sep_node_islist` (introduced at line 10). This lemma is analogous to the $\star$ used in the definition of `islist` in separation logic. To account for the $\star$ in the loop invariant, we first introduce a new separation predicate `sep_islist_islist` (line 14) and then we use it in the loop invariant (line 21).

With these annotations, the axiomatizations and the definitions automatically generated for `sep_node_islist` and `sep_islist_islist` allow a general-purpose SMT solver such as Alt-Ergo or CVC3 to discharge all verification conditions obtained by weakest precondition for the code in Fig. 1, in no time.

3 Separation Predicates

3.1 Inductive Definitions

A separation predicate is generated from user-defined inductive predicates. The generation is sound only if the definitions of the inductive predicates obey several constraints, the main one being that two distinct cases should not overlap. Fortunately, this is the case for most common inductive predicates. For instance, predicate `islist` from Fig. 1 (lines 4–8) trivially satisfies the non-overlapping constraint, since `p` cannot be both null and non-null.

Generally speaking, we consider inductive definitions following the syntax given in Fig. 2. The constraints are then the following:

```
1  struct node { int hd; struct node *next; };
2
3  /*@
4  inductive islist(struct node *p) {
5   case nil: islist(\null);
6   case cons: \forall struct node *p; p != \null ==> \valid(p) ==>
7   islist(p->next) ==> islist(p);
8  }
9
10 #Gen_Separation sep_node_islist(struct node*, islist)
11 lemma list_sep:
12   \forall struct node *p; p!=null ==>
13     islist(p) ==> sep_node_islist(p, p->next);
14
15 #Gen_Separation sep_islist_islist(islist, islist)
16 @*/
17
18 /*@ requires islist(p); ensures islist(\result); @*/
19 struct node * rev(struct node *p) {
20   struct node *q = NULL;
21   /*@ loop invariant
22         islist(p) && islist(q) && sep_islist_islist(p,q); @*/
23   while(p != NULL) {
24     struct node *tmp = p->next;
25     p->next = q;
26     q = p;
27     p = tmp;
28   }
29   return q;
30 }
```

Fig. 1. List Reversal.

$$
\begin{array}{rrl}
\text{(terms)} & t & ::= \quad x \mid t{\rightarrow}\texttt{field} \mid \phi(\boldsymbol{t}) \\
\text{(formulas)} & f & ::= \quad t = t \mid \neg(t = t) \mid p(\boldsymbol{x}) \\
\text{(inductive case)} & c & ::= \quad \texttt{C} : \forall \boldsymbol{x}. f \Rightarrow \ldots \Rightarrow f \Rightarrow p(\boldsymbol{x}) \\
\text{(inductive definition)} & d & ::= \quad \texttt{inductive}\, p(\boldsymbol{x}) = c \mid \ldots \mid c
\end{array}
$$

Fig. 2. Inductive Definitions.

- in a term t, a function symbol ϕ cannot refer to the memory state;
- in a formula f, a predicate symbol p can refer to the memory state only if it is an inductively defined predicate following the constraints (which includes the predicate being defined);
- if $\texttt{C}_\texttt{i} : \forall \boldsymbol{x}. f_{i,1} \Rightarrow \ldots \Rightarrow f_{i,n_i} \Rightarrow p(\boldsymbol{x})$ and $\texttt{C}_\texttt{j} : \forall \boldsymbol{x}. f_{j,1} \Rightarrow \ldots \Rightarrow f_{j,n_j} \Rightarrow p(\boldsymbol{x})$ are two distinct cases of $\texttt{inductive}\, p(\boldsymbol{x}) = c_1 \mid \ldots \mid c_n$, then we should have

$$\forall \boldsymbol{x}. \neg(f_{i,1} \wedge \cdots \wedge f_{i,n_i} \wedge f_{j,1} \wedge \cdots \wedge f_{j,n_j}).$$

It is worth pointing out that an inductive predicate which is never used to define a separation predicate does not have to follow these restrictions.

3.2 An Axiomatization of Footprints

The footprint of an inductive predicate p is the set of pointers which it depends on. More precisely, in a memory state m where $p(\boldsymbol{x})$ is true, the pointer q is in the footprint of $p(\boldsymbol{x})$ if we can modify the value q points at such that $p(\boldsymbol{x})$ does not hold anymore. Such a definition is too precise to be used in practice. We use instead a coarser notion of footprint, which is derived from the definition of p and over-approximates the precise footprint.

Let us consider the definition of `islist`. First, we introduce a new type `ft` for footprints. Then we declare a function symbol $\texttt{ft}_{\texttt{islist}}$ and a predicate symbol $\in$. The intended semantics is the following: $\texttt{ft}_{\texttt{islist}}(m, p)$ is the footprint of $\texttt{islist}(p)$ in memory state m and $q \in \texttt{ft}_{\texttt{islist}}(m, p)$ means that q belongs to the footprint $\texttt{ft}_{\texttt{islist}}(m, p)$. Both symbols are axiomatized simultaneously as follows:

$$\forall q. \forall m. \forall p.\, q \in \texttt{ft}_{\texttt{islist}}(m, p) \Leftrightarrow \left(\begin{array}{c} p \neq \texttt{NULL} \wedge \texttt{islist}(m, \{p{\rightarrow}\texttt{next}\}_m) \\ \wedge (q = p \vee q \in \texttt{ft}_{\texttt{islist}}(m, \{p{\rightarrow}\texttt{next}\}_m)) \end{array} \right)$$

where $\{p{\rightarrow}\texttt{next}\}_m$ stands for expression $p{\rightarrow}\texttt{next}$ in memory state m.

Then separation predicates are easily defined from footprints. The pragma from line 10 in Fig. 1 generates the definition

$$\texttt{sep\_node\_islist}(m, q, p) \triangleq q \notin \texttt{ft}_{\texttt{islist}}(m, p)$$

and pragma from line 14 generates the definition

$$\begin{array}{l} \texttt{sep\_islist\_islist}(m, p_1, p_2) \triangleq \\ \quad \forall q.\, q \notin \texttt{ft}_{\texttt{islist}}(m, p_1) \vee q \notin \texttt{ft}_{\texttt{islist}}(m, p_2) \end{array}$$

(where $q \notin s$ stands for $\neg(q \in s)$). The predicate symbols and the types that appears in the pragma specify the signature of the separation predicate and which inductive predicate must be used to defined the separation predicate. A type is viewed as the predicate symbol of an unary predicate of this type whose footprint is reduced to its argument. The signature of the defined separation predicate is the concatenation of the signature of the predicate symbols.

Generally speaking, in order to axiomatize the footprint of an inductive predicate, we first introduce a meta-operation $\mathtt{FT}_{m,q}(e)$ that builds a formula expressing that q is in the footprint of a given expression e in memory state m:

$$\begin{aligned}
\mathtt{FT}_{m,q}(x) &= \bot \\
\mathtt{FT}_{m,q}(t{\rightarrow}\mathtt{j}) &= \mathtt{FT}_{m,q}(t) \vee q = t \\
\mathtt{FT}_{m,q}(\phi(\boldsymbol{t})) &= \bigvee_j \mathtt{FT}_{m,q}(t_j) \\
\mathtt{FT}_{m,q}(t_1 = t_2) &= \mathtt{FT}_{m,q}(\neg(t_1 = t_2)) = \mathtt{FT}_{m,q}(t_1) \vee \mathtt{FT}_{m,q}(t_2) \\
\mathtt{FT}_{m,q}(p(\boldsymbol{t})) &= \bigvee_j \mathtt{FT}_{m,q}(t_j) \vee q \in \mathtt{ft}_p(m, \boldsymbol{t})
\end{aligned}$$

We pose $q \in \mathtt{ft}_p(m, \boldsymbol{t}) \triangleq \bot$ whenever predicate p does not depend on the memory state. Then the footprint of an inductive predicate p defined by $\mathtt{inductive}\, p(\boldsymbol{x}) = c_1 | \ldots | c_n$ with c_i being $\mathtt{C_i} : \forall \boldsymbol{x}. f_{i,1} \Rightarrow \ldots \Rightarrow f_{i,n_i} \Rightarrow p(\boldsymbol{x})$ is axiomatized as follows:

$$\forall q. \forall m. \forall \mathbf{x}.\, q \in \mathtt{ft}_p(m, \mathbf{x}) \Leftrightarrow \bigvee_i \left(\bigwedge_j \overline{f_{i,j}} \wedge \bigvee_j \mathtt{FT}_{m,q}(f_{i,j}) \right)$$

where $\overline{f_{i,j}}$ is the version of $f_{i,j}$ with the memory explicited (eg. $\overline{t{\rightarrow}\mathtt{j}} = \{t{\rightarrow}\mathtt{j}\}_m$). In the axiom above for the footprint of `islist`, we simplified the `NULL` case since it is equivalent to $\bot$.

With the footprints of the inductive predicates you can now define the separation predicate. A separation predicate that define the separation of n inductive predicates is defined as the conjunction of all the disjunction $q \in \mathtt{ft}_{p_i}(m, \boldsymbol{x_i}) \vee q \in \mathtt{ft}_{p_j}(m, \boldsymbol{x_j})$ between the footprint of the inductive predicate. The soundness of this construction have been proved in [3].

The separation predicates allow you to translate a large set of separation logic formulas, namely first-order separation logic formula without magic wand and with separation conjunction used only on inductive predicates which definitions satisfy our constraints.

3.3 Mutation Axioms

The last ingredient we generate is a mutation axiom. It states the main property of the footprint, namely that an assignment outside the footprint does not invalidate the corresponding predicate. In the case of `islist`, the mutation axiom is

$$\forall m, p, q, v.\, q \notin \mathtt{ft}_{\mathtt{islist}}(m, p) \Rightarrow \mathtt{islist}(m, p) \Rightarrow \mathtt{islist}(m[q{\rightarrow}\mathtt{next} := v], p)$$

where $m[q{\rightarrow}\,\texttt{next} := v]$ stands for a new memory state obtained from m by assigning value v to memory location $q{\rightarrow}\texttt{next}$. Actually, this property could be proved from the definition of $\texttt{ft}_{\texttt{islist}}$, but this would require induction. Since this is out of reach of SMT solvers, we state it as an axiom. We do not require the user to discharge it as a lemma, since it is proved sound in the meta-theory [3]. This is somehow analogous to the mutation rule of separation logic, which is proved sound in the meta-theory. The mutation rule of separation logic also allows proving that two formulas stay separated if you modify something separated from both of them. We can prove the same by adding an autoframe axiom, which is reminiscent of the autoframe concept in dynamic frames [16]:

$$\begin{aligned}\forall m, p, q, v.\ q \notin \texttt{ft}_{\texttt{islist}}(m, p) \Rightarrow \texttt{islist}(m, p) \Rightarrow \\ \texttt{ft}_{\texttt{islist}}(m, p) = \texttt{ft}_{\texttt{islist}}(m[q{\rightarrow}\texttt{next} := v], p)\end{aligned}$$

Generally speaking, for each inductive predicate p and for each field `field` we add the following axioms :

$$\forall q.\forall v.\forall m.\forall \boldsymbol{x}.\neg q \in \texttt{ft}_p(m, \boldsymbol{x}) \Rightarrow p(m, \boldsymbol{x}) \Rightarrow p(m[q{\rightarrow}\texttt{field} := v], \boldsymbol{x})$$

and

$$\begin{aligned}\forall q.\forall v.\forall m.\forall \boldsymbol{x}.\ \neg q \in \texttt{ft}^c_p(m, \boldsymbol{x}) \Rightarrow p(m, \boldsymbol{x}) \Rightarrow \\ \texttt{ft}_p(m, \mathtt{x}) = \texttt{ft}_p(m[q{\rightarrow}\texttt{field} := v], \boldsymbol{x}).\end{aligned}$$

The distinctness of the cases of the inductive predicate p appears in the proof of the autoframe property.

3.4 Implementation

Our generation of separation predicates is implemented in the Frama-C/Jessie tool chain for the verification of C programs [11, 21, 5]. This tool chain can be depicted as follows:

From a technical point of view, our implementation is located in the Jessie tool, since this is the first place where the memory model is made explicit[1]. Jessie uses the component-as-array model also known as the Burstall-Bornat memory model [7, 6]. Each structure field is modeled using a distinct applicative array. Consequently, function and predicate symbols such as $\texttt{ft}_{\texttt{islist}}$ or `islist` do not take a single argument m to denote memory state, but one or several applicative arrays instead, one for each field mentioned in the inductive definition. Similarly, a quantification $\forall m$ in our meta-theory (Sec. 3.2 and 3.3 above) is materialized in the implementation by one or several quantifications over applicative arrays, one

[1] Since we could not extend the ACSL language with the new pragmas for separation, we have to modify the Jessie input file manually at each run. Furthermore we use in the assigns clauses the keyword `\all` that does not exist yet in ACSL

for each field appearing in the formula. In the case of `islist`, for instance, quantification $\forall m$ becomes $\forall$`next`, expression $\{p{\rightarrow}\texttt{next}\}_m$ becomes `get`$(\texttt{next}, p)$, and expression $m[q{\rightarrow}\texttt{next} := v]$ becomes `set`$(\texttt{next}, p, v)$, where `get` and `set` are access and update operations over applicative arrays. Additionally, we have to define one footprint symbol for each field.

It is worth pointing out that we made no modification at all in Why3 to support our separation predicates. Only Jessie has been modified.

4 A Case Study: Composite Pattern

To show the usefulness of separation predicates, we consider the problem of verifying an instance of the *Composite Pattern*, as proposed in the VACID-0 benchmark [20].

4.1 The Problem

We consider a forest, whose nodes are linked upward through parent links. Each node carries an integer value, as well as the sum of the values for all nodes in its subtree (including itself). The corresponding C structure is thus defined as follows:

```
struct node {
  int val, sum;
  struct node *parent;
};
typedef struct node *NODE;
```

The operations considered here are the following: `NODE create(int v);`, creates a new node; `void update(NODE p, int v);`, assigns value `v` to node `p`; `void addChild(NODE p, NODE q);`, set node `p` as `q`'s parent, assuming node `q` has no parent; `void dislodge(NODE p);`, disconnects `p` from its parent, if any.

One challenge with such a data structure is that operations `update`, `addChild`, and `dislodge` have non-local consequences, as the `sum` field must be updated for all ancestors. Another challenge is to prevent `addChild` from creating a cycle, *i.e.*, to express that node `q` is not already an ancestor of node `p`. Thus we prove the memory safety and the correct behavior of these operations.

4.2 Code and Specification

Our annotated C code for this instance of the composite pattern is given in the appendix. In this section, we comment on the key aspects of our solution. The annotations are written in the ACSL specification language. The behavior of the functions are defined by contract: the keyword `requires` introduces the precondition expressed by a first-order formula, the keyword `ensures` introduces the post-conditions, and the keyword `assigns` introduces the set of memory location that can be modified by a call to the function. The precondition and

this set are interpreted before the execution of the function, the post-conditions is interpreted after. One can refer in the post-condition to the state before the execution of the function using the keyword `\old`. It must be remarked that if a field of a type is never modified in the body of a function you don't need to mention it in the assigns clauses. Moreover the component-as-array memory model ensures without reasoning that any formulas that depend only of such fields remain true after a call to the function.

Separation Predicate. For the purpose of `addChild`'s specification, we use a separation predicate. It states that a given node is disjoint from the list of ancestors of another node. Such a list is defined using predicate `parents` (lines 7–12), which is similar to predicate `islist` in the previous section. The separation predicate, `sep_node_parents`, is then introduced on line 14 and used in the precondition of `addChild` on line 84.

This is a crucial step, since otherwise assignment `q->parent = p` on line 95 could break property `parents(p)`. Such a property is indeed required by `upd_inv` to ensure its termination.

Restoring the Invariant. As suggested in VACID-0 [20], we introduce a function to restore the invariant (function `upd_inv` on lines 68–77). Given a node `p` and an offset `delta`, it adds `delta` to the `sum` field of `p` and of all its ancestors. This way, we reuse this function in `addChild` (with the new child's sum), in `update` (with the difference), and in `dislodge` (with the opposite of the child's sum).

Local and Global Invariant. Another key ingredient of the proof is to ensure the invariant property that, for each node, the `sum` field contains the sum of values in all nodes beneath, including itself. To state such a property, we need to access children nodes. Since the data structure does not provide any way to do that (we only have parent links), we augment the data structure with ghost children links. To make it simple, we assume that each node has at most two children, stored in ghost fields `left` and `right` (line 4). Structural invariants relating fields `parent`, `left`, and `right` are gathered in predicate `wf` (lines 28–37).

To state the invariant for `sum` fields, we first introduce a predicate `good` (lines 20–23). It states that the `sum` field of a given node `p` has a correct value when `delta` is added to it. It is important to notice that predicate `good` is a *local* invariant, which assumes that the left and right children of `p` have correct sums. Then we introduce a predicate `inv` (lines 25–26) to state that any node `p` verifies `good(p, 0)`, with the possible exception of node `except`. Using an exception is convenient to state that the invariant is locally violated during `upd_inv`. To state that the invariant holds for all nodes, we simply use `inv(NULL)`.

Our local invariant is convenient, as it does not require any induction. However, to convince the reader that we indeed proved the expected property, we also show that this local invariant implies a global, inductively-defined invariant. Lines 130–137 introduce the sum of all values in a tree, as an inductive predicate `treesum`, and a lemma to state that local invariant `inv(NULL)` implies $\texttt{treesum}(p, p{\rightarrow}\texttt{sum})$ for any node p.

4.3 Proof

The proof was performed using Frama-C Carbon[2] and its Jessie plug-in [21], using SMT solvers Alt-Ergo 0.92.3, CVC3 2.2, and Z3 2.19, on an Intel Core Duo 2.4 GHz. As explained in Sec. 3.4, we first run Frama-C on the annotated C code and then we insert the separation pragmas in the generated Jessie code (this is a benign modification). All verification conditions are discharged automatically within a total time of 30 seconds.

The two lemmas `parents_sep` and `global_invariant` were proved interactively using the Coq proof assistant version 8.3pl3 [26]. A total of 100 lines of tactics is needed. It doesn't take more than three days for one of the author to find the good specifications and make the proofs.

5 Function Footprints

In the case of the composite pattern, it is easy to specify the footprints of the C functions. Indeed, we can simply say that any `sum` field may be modified (using `\all->sum` in `assigns` clauses), since the invariant provides all necessary information regarding the contents of `sum` fields. For a function such as list reversal, however, we need to be more precise. We want to know that any list separated from the one being reversed is left unmodified. For instance, we would like to be able to prove the following piece of code:

```
1  /*@
2  requires islist(p) && islist(q) && sep_list_list(p,q);
3  ensures  islist(p) && islist(q) && sep_list_list(p,q);
4  @*/
5  void bar(struct node * p, struct node * q) {
6    p = rev(p);
7  }
```

For that purpose we must strengthen the specification and loop invariant of function `rev` with a suitable frame property. One possibility is to proceed as follows:

```
1  /*@
2  #Gen_Frame: list_frame list
3  #Gen_Sub: list_sub list list
4
5  requires list(p);
6  ensures  list(\result) && list_frame{Old,Here}(p,result);
7  @*/
8  struct node * rev(struct node * p);
9    ...
10   /*@ loop invariant
```

[2] `http://frama-c.com/`

```
11          list(p) &&& list(q) &&& sep_list_list(p,q)
12          &&& list_frame{Init,Here}(\at(p,Init),q)
13          &&& list_sub{Init,Here}(\at(p,Init),p); @*/
14  ...
```

Two pragmas introduce new predicates `list_frame` and `list_sub`. Both depend on two memory states. The formula `list_frame{Old,Here}(p,result)` expresses in the post-condition that, between pre-state `Old` and post-state `Here`, all modified pointers belong to list `p`. It also specifies that the footprint of list `result` is included in the (old) footprint of list `p`. On the example of function `bar`, we now know that only pointers from `p` have been modified, so we can conclude that `islist(q)` is preserved. Additionally, we know that the footprint of `islist(p)` has not grown so we can conclude that it is still separated from `islist(q)`. The formula `list_sub{Init,Here}(\at(p,Init),p)` specifies only the inclusion of the footprint of the lists.

These two predicates could be axiomatized using membership only. For instance, $\texttt{list\_sub}(p, q)$ could be simply axiomatized as $\forall x, x \in \texttt{ft}_{\texttt{islist}}(p) \Rightarrow x \in \texttt{ft}_{\texttt{islist}}(q)$. But doing so has a rather negative impact on SMT solvers, as they have to first instantiate this axiom and then to resort to other axioms related to membership. Moreover this axiom is very generic and can be applied when not needed. For that reason we provide, in addition to axioms related to membership, axioms for footprint inclusion, to prove either $s \subset \texttt{ft}_p(p)$ or $\texttt{ft}_p(p) \subset s$ directly. With such axioms, functions `rev` and `bar` are proved correct automatically.

6 Related and Future Work

VeriFast [13, 15] allows user-defined predicates but requires user annotations to fold or unfold these predicates. In our work, we rely instead on the capability of first-order provers to fold and unfold definitions. VeriFast uses the SMT solver Z3, but only as a constraint solver on ground terms.

The technique of *implicit dynamic frames* [24] is closer to our work, except that formulas are restricted. Additionally, implicit dynamic frames make use of a set theory, whereas we do not require any, as we directly encode the relevant parts of set theory inside our footprint definition axioms.

Both these works do not allow a function to access (and thus modify) a pointer that is not in the footprint of the function's precondition — except if it is allocated inside the function. In our work, we do not have such a restriction. When necessary, we may define the footprint of a function using separation predicates, as explained in the first author's thesis [3].

There exist already several proofs of the composite pattern. One is performed using VeriFast [14]. It requires many lemmas and many `open`/`close` statements, whereas our proof does not contain much proof-related annotations.

The use of a local invariant in our proof is not new. It was first described in [19]. The proof by Rosenberg, Banerjee, and Naumann [23] also makes use of it. In order to prove that `addChild` is not creating cycles, the latter proof

introduces two ghost fields, one for the set of descendants and one for the root node of the tree. Updating these ghost fields must be done at several places. In our case, we could manage to perform the case only with the generated predicate `sep_node_parents` without need of extra ghost fields which leads to a simpler proof.

The composite pattern has also been proved using *considerate reasoning* [25], a technique that advocates for local invariant like the one we used. Our predicate `inv` is similar to their `broken` declaration. As far as we understand, this proof is not mechanized, though.

Our future work includes generalizing the frame pragma used to describe the footprint of a function. One solution is to compute the footprint directly from ACSL's `assigns` clause, if any. Another is to describe the footprint using the linear maps framework [18]. One valuable future work would be to formally prove the consistency of our axioms, either using a meta-theoretical formalization, or, in a more tractable way, by producing proofs for each generated axiom.

References

1. Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302, Berlin, Germany, July 2007. Springer.
2. Josh Berdine, Cristiano Calcagno, and Peter W. O'hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *In International Symposium on Formal Methods for Components and Objects*, pages 115–137. Springer, 2005.
3. François Bobot. *Logique de séparation et vérification déductive*. Thèse de doctorat, Université Paris-Sud, December 2011.
4. François Bobot, Sylvain Conchon, Évelyne Contejean, Mohamed Iguernelala, Stéphane Lescuyer, and Alain Mebsout. The Alt-Ergo automated theorem prover, 2008. `http://alt-ergo.lri.fr/`.
5. François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, Wrocław, Poland, August 2011.
6. Richard Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.
7. Rod Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.
8. Leonardo de Moura and Nikolaj Bjørner. Z3, an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
9. Leonardo de Moura and Bruno Dutertre. Yices: An SMT Solver. `http://yices.csl.sri.com/`.
10. Jean-Christophe Filliâtre, Andrei Paskevich, and Aaron Stump. The 2nd Verified Software Competition, November 2011. `https://sites.google.com/site/vstte2012/compet`.
11. The Frama-C platform for static analysis of C programs, 2008. `http://www.frama-c.cea.fr/`.
12. Marieke Huisman, Vladimir Klebanov, and Rosemary Monahan, October 2011. `http://foveoos2011.cost-ic0701.org/verification-competition`.

13. Bart Jacobs and Frank Piessens. The verifast program verifier. CW Reports CW520, Department of Computer Science, K.U.Leuven, August 2008.
14. Bart Jacobs, Jan Smans, and Frank Piessens. Verifying the composite pattern using separation logic. In *Workshop on Specification and Verification of Component-Based Systems, Challenge Problem Track*, November 2008.
15. Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the verifast program verifier. In *Programming Languages and Systems (APLAS 2010)*, pages 304–311. Springer-Verlag, November 2010.
16. Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *14th International Symposium on Formal Methods (FM'06)*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283, Hamilton, Canada, 2006.
17. Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T. Leavens, Valentin Wüstholz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark Hillebrand, Bart Jacobs, K. Rustan M. Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan Tobies, Thomas Tuerk, Mattias Ulbrich, and Benjamin Weiß. The 1st Verified Software Competition: Experience report. In Michael Butler and Wolfram Schulte, editors, *Proceedings, 17th International Symposium on Formal Methods (FM 2011)*, volume 6664 of *LNCS*. Springer, 2011. Materials available at `www.vscomp.org`.
18. Shuvendu K. Lahiri, Shaz Qadeer, and David Walker. Linear maps. In *Proceedings of the 5th ACM workshop on Programming languages meets program verification*, PLPV '11, pages 3–14, New York, NY, USA, 2011. ACM.
19. Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 2007.
20. K. Rustan M. Leino and Michał Moskal. VACID-0: Verification of ample correctness of invariants of data-structures, edition 0. In *Proceedings of Tools and Experiments Workshop at VSTTE*, 2010.
21. Yannick Moy and Claude Marché. *The Jessie plugin for Deduction Verification in Frama-C — Tutorial and Reference Manual*. INRIA & LRI, 2011. `http://krakatoa.lri.fr/`.
22. J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *17h Annual IEEE Symposium on Logic in Computer Science*. IEEE Comp. Soc. Press, 2002.
23. Stan Rosenberg, Anindya Banerjee, and David A. Naumann. Local reasoning and dynamic framing for the composite pattern and its clients. In Gary T. Leavens, Peter W. O'Hearn, and Sriram K. Rajamani, editors, *VSTTE*, volume 6217 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2010.
24. Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In Sophia Drossopoulou, editor, *ECOOP 2009 — Object-Oriented Programming*, Lecture Notes in Computer Science, pages 148–172. Springer Berlin / Heidelberg, 2009.
25. Alexander J. Summers and Sophia Drossopoulou. Considerate reasoning and the composite design pattern. In Gilles Barthe and Manuel V. Hermenegildo, editors, *VMCAI*, volume 5944 of *Lecture Notes in Computer Science*, pages 328–344. Springer, 2010.
26. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.3*, 2010. `http://coq.inria.fr`.

A Annotated Source Code

```
1  typedef struct node {
2    int val, sum;
3    struct node *parent;
4    //@ ghost struct node *left, *right;
5  } *NODE;
6
7  /*@ inductive parents(NODE p) {
8    case nil: \forall NODE p; p==NULL ==> parents(p);
9    case cons: \forall NODE p;
10     p != NULL ==> \valid(p) ==>
11     parents(p->parent) ==> parents(p);
12 }
13
14 #Gen_Separation sep_node_parents(NODE, parents)
15
16 lemma parents_sep:
17   \forall NODE p; p!=NULL ==>
18     parents(p) ==> sep_node_parents(p, p->parent);
19
20 predicate good(NODE p, int delta) =
21   p->sum + delta == p->val +
22    (p->left == NULL? 0 : p->left->sum) +
23    (p->right == NULL? 0 : p->right->sum);
24
25 predicate inv(NODE except) =
26   \forall NODE p; \valid(p) ==> p != except ==> good(p, 0);
27
28 predicate wf(NODE except) =
29   \forall NODE p; \valid(p) ==> p != except ==>
30     (p->right != NULL ==>
31        p->right->parent == p && \valid(p->right)) &&
32     (p->left != NULL ==>
33        p->left->parent == p && \valid(p->left)) &&
34     (p->right == p->left ==> p->right == NULL) &&
35     (p->parent != NULL ==> \valid(p->parent)) &&
36     (p->parent != NULL ==>
37        p->parent->left == p || p->parent->right == p);
38
39 predicate newnode(NODE p, integer v) =
40        parents(p) && p->right == NULL && p->left == NULL &&
41        p->parent == NULL && p->val == v && \valid(p);
42 @*/
43
44 /*@ requires
45       inv(NULL) && wf(NULL);
46     ensures
47       inv(NULL) && wf(NULL) && newnode(\result, v) &&
```

```
48        \forall NODE n; \old(\valid(n)) ==>
49           \result != n && \valid(n) &&
50           \old(n->val) == n->val && \old(n->parent) == n->parent &&
51           \old(n->left) == n->left && \old(n->right) == n->right;
52  @*/
53  NODE create(int v) {
54   Before:
55     {
56    NODE p = (NODE)malloc(sizeof(struct node));
57    /*@ assert \forall NODE n; n != p ==>
58                \valid(n) ==> \at(\valid(n),Before); @*/
59    p->val = p->sum = v;
60    p->parent = p->left = p->right = NULL;
61    return p;
62  }}
63
64  /*@ requires inv(p) && parents(p) && wf(NULL) && good(p,delta);
65      ensures inv(NULL);
66      assigns \all->sum;
67   @*/
68  void upd_inv(NODE p, int delta) {
69     NODE n = p;
70     /*@ loop invariant
71           inv(n) && parents(n) && (n != NULL ==> good(n,delta));
72      @*/
73      while (n != NULL) {
74        n->sum = n->sum + delta;
75        n = n->parent;
76      }
77  };
78
79  /*@
80    requires
81      inv(NULL) && wf(NULL) &&
82      \valid(q) && q->parent == NULL &&
83      parents(p) && p != NULL && sep_node_parents(p, p->parent) &&
84      (p->left == NULL || p->right == NULL) && sep_node_parents(q,p);
85    ensures
86      parents(q) && parents(p) && inv(NULL) && wf(NULL) &&
87      (\old(p->left) == NULL ==>
88         p->left == q && \old(p->right) == p->right) &&
89      (\old(p->left) != NULL ==>
90         p->right == q && \old(p->left) == p->left);
91    assigns p->left, p->right, q->parent, \all->sum;
92  @*/
93  void addChild(NODE p, NODE q) {
94    if (p->left == NULL) p->left = q; else p->right = q;
95    q->parent = p;
96    upd_inv(p, q->sum);
97  }
```

```
98
99  /*@ requires parents(p) &&& p != NULL &&& inv(NULL) &&& wf(NULL);
100     ensures p->val == v &&& parents(p) &&& inv(NULL) &&& wf(NULL);
101     assigns p->val, \all->sum;
102 @*/
103 void update(NODE p, int v) {
104    int delta = v - p->val;
105    p->val = v;
106    upd_inv(p, delta);
107 }
108
109 /*@
110   requires
111     parents(p) &&& p != NULL &&& p->parent != NULL &&&
112     inv(NULL) &&& wf(NULL);
113   ensures
114     parents(p) &&& p->parent == NULL &&& inv(NULL) &&& wf(NULL) &&&
115     (\old(p->parent->left) == p ==>
116        \old(p->parent)->left == NULL) &&&
117     (\old(p->parent->right) == p ==>
118        \old(p->parent)->right == NULL);
119   assigns p->parent->left, p->parent->right, p->parent, \all->sum;
120 @*/
121 void dislodge(NODE p) {
122   NODE n = p->parent;
123   if(p->parent->left == p) p->parent->left = NULL;
124   if(p->parent->right == p) p->parent->right = NULL;
125   p->parent = NULL;
126   upd_inv(n, -p->sum);
127 }
128
129 /*@
130 inductive treesum{L}(NODE p, integer v) {
131   case treesum_null{L}:
132     treesum(NULL, 0);
133   case treesum_node{L}:
134     \forall NODE p; p != NULL ==> \forall integer sl, sr;
135     treesum(p->left, sl) ==> treesum(p->right, sr) ==>
136     treesum(p, p->val + sl + sr);
137 }
138
139 lemma global_invariant{L}:
140   inv(NULL) ==> wf(NULL) ==>
141   \forall NODE p; \valid(p) ==> treesum(p, p->sum);
142 @*/
```